Chapter 5

# Hash Tables

Hash tables are an efficient method of storing a small number, n, of integers from a large range $U = \{0, \ldots, 2^w - 1\}$. The term *hash table* includes a broad range of data structures. This chapter focuses on one of the most common implementations of hash tables, namely hashing with chaining.

Very often hash tables store types of data that are not integers. In this case, an integer *hash code* is associated with each data item and is used in the hash table. The second part of this chapter discusses how such hash codes are generated.

Some of the methods used in this chapter require random choices of integers in some specific range. In the code samples, some of these "random" integers are hard-coded constants. These constants were obtained using random bits generated from atmospheric noise.

## 5.1 ChainedHashTable: Hashing with Chaining

A ChainedHashTable data structure uses *hashing with chaining* to store data as an array, t, of lists. An integer, n, keeps track of the total number of items in all lists (see Figure 5.1):

ChainedHashTable

```
List<T>[] t;
int n;
```

The *hash value* of a data item x, denoted hash(x) is a value in the range

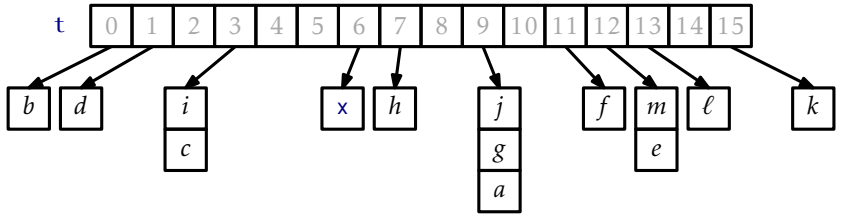


Figure 5.1: An example of a `ChainedHashTable` with `n` = 14 and `t.length` = 16. In this example `hash(x)` = 6

{0,...,`t.length` − 1}. All items with hash value `i` are stored in the list at `t[i]`. To ensure that lists don't get too long, we maintain the invariant

$$n \le t.length$$

so that the average number of elements stored in one of these lists is $n/t.length \le 1$.

To add an element, `x`, to the hash table, we first check if the length of `t` needs to be increased and, if so, we grow `t`. With this out of the way we hash `x` to get an integer, `i`, in the range {0,...,`t.length` − 1}, and we append `x` to the list `t[i]`:

ChainedHashTable
```
boolean add(T x) {
  if (find(x) != null) return false;
  if (n+1 > t.length) resize();
  t[hash(x)].add(x);
  n++;
  return true;
}
```

Growing the table, if necessary, involves doubling the length of `t` and reinserting all elements into the new table. This strategy is exactly the same as the one used in the implementation of `ArrayStack` and the same result applies: The cost of growing is only constant when amortized over a sequence of insertions (see Lemma 2.1 on page 33).

Besides growing, the only other work done when adding a new value `x` to a `ChainedHashTable` involves appending `x` to the list `t[hash(x)]`. For

any of the list implementations described in Chapters 2 or 3, this takes only constant time.

To remove an element, `x`, from the hash table, we iterate over the list `t[hash(x)]` until we find `x` so that we can remove it:

ChainedHashTable
```
T remove(T x) {
  Iterator<T> it = t[hash(x)].iterator();
  while (it.hasNext()) {
    T y = it.next();
    if (y.equals(x)) {
      it.remove();
      n--;
      return y;
    }
  }
  return null;
}
```

This takes $O(n_{\texttt{hash(x)}})$ time, where $n_i$ denotes the length of the list stored at `t[i]`.

Searching for the element `x` in a hash table is similar. We perform a linear search on the list `t[hash(x)]`:

ChainedHashTable
```
T find(Object x) {
  for (T y : t[hash(x)])
    if (y.equals(x))
      return y;
  return null;
}
```

Again, this takes time proportional to the length of the list `t[hash(x)]`.

The performance of a hash table depends critically on the choice of the hash function. A good hash function will spread the elements evenly among the `t.length` lists, so that the expected size of the list `t[hash(x)]` is $O(\texttt{n}/\texttt{t.length}) = O(1)$. On the other hand, a bad hash function will hash all values (including `x`) to the same table location, in which case the size

of the list t[hash(x)] will be n. In the next section we describe a good hash function.

### 5.1.1 Multiplicative Hashing

Multiplicative hashing is an efficient method of generating hash values based on modular arithmetic (discussed in Section 2.3) and integer division. It uses the div operator, which calculates the integral part of a quotient, while discarding the remainder. Formally, for any integers $a \geq 0$ and $b \geq 1$, $a \operatorname{div} b = \lfloor a/b \rfloor$.

In multiplicative hashing, we use a hash table of size $2^d$ for some integer d (called the *dimension*). The formula for hashing an integer $x \in \{0, \ldots, 2^w - 1\}$ is

$$\mathtt{hash}(x) = ((z \cdot x) \bmod 2^w) \operatorname{div} 2^{w-d} \ .$$

Here, z is a randomly chosen *odd* integer in $\{1, \ldots, 2^w - 1\}$. This hash function can be realized very efficiently by observing that, by default, operations on integers are already done modulo $2^w$ where w is the number of bits in an integer. (See Figure 5.2.) Furthermore, integer division by $2^{w-d}$ is equivalent to dropping the rightmost $w - d$ bits in a binary representation (which is implemented by shifting the bits right by $w-d$). In this way, the code that implements the above formula is simpler than the formula itself:

ChainedHashTable

```
int hash(Object x) {
  return (z * x.hashCode()) >>> (w-d);
}
```

The following lemma, whose proof is deferred until later in this section, shows that multiplicative hashing does a good job of avoiding collisions:

**Lemma 5.1.** *Let* x *and* y *be any two values in* $\{0, \ldots, 2^w - 1\}$ *with* $x \neq y$. *Then* $\Pr\{\mathtt{hash}(x) = \mathtt{hash}(y)\} \leq 2/2^d$.

With Lemma 5.1, the performance of remove(x), and find(x) are easy to analyze:

| | |
|---|---|
| $2^w$ (4294967296) | 10000000000000000000000000000000 |
| $z$ (4102541685) | 11110100100001111101000101110101 |
| $x$ (42) | 00000000000000000000000000101010 |
| $z \cdot x$ | 10100000011110010010000101110100110010 |
| $(z \cdot x) \bmod 2^w$ | 00011110010010000101110100110010 |
| $((z \cdot x) \bmod 2^w) \operatorname{div} 2^{w-d}$ | 00011110 |

Figure 5.2: The operation of the multiplicative hash function with $w = 32$ and $d = 8$.

**Lemma 5.2.** *For any data value $x$, the expected length of the list $t[\texttt{hash}(x)]$ is at most $n_x + 2$, where $n_x$ is the number of occurrences of $x$ in the hash table.*

*Proof.* Let $S$ be the (multi-)set of elements stored in the hash table that are not equal to $x$. For an element $y \in S$, define the indicator variable

$$I_y = \begin{cases} 1 & \text{if } \texttt{hash}(x) = \texttt{hash}(y) \\ 0 & \text{otherwise} \end{cases}$$

and notice that, by Lemma 5.1, $\mathrm{E}[I_y] \le 2/2^d = 2/t.\texttt{length}$. The expected length of the list $t[\texttt{hash}(x)]$ is given by

$$\begin{aligned} \mathrm{E}[t[\texttt{hash}(x)].\texttt{size}()] &= \mathrm{E}\left[n_x + \sum_{y \in S} I_y\right] \\ &= n_x + \sum_{y \in S} \mathrm{E}[I_y] \\ &\le n_x + \sum_{y \in S} 2/t.\texttt{length} \\ &\le n_x + \sum_{y \in S} 2/n \\ &\le n_x + (n - n_x)2/n \\ &\le n_x + 2 \ , \end{aligned}$$

as required. □

Now, we want to prove Lemma 5.1, but first we need a result from number theory. In the following proof, we use the notation $(b_r, \ldots, b_0)_2$ to denote $\sum_{i=0}^{r} b_i 2^i$, where each $b_i$ is a bit, either 0 or 1. In other words,

$(b_r,\ldots,b_0)_2$ is the integer whose binary representation is given by $b_r,\ldots,b_0$. We use $\star$ to denote a bit of unknown value.

**Lemma 5.3.** *Let $S$ be the set of odd integers in $\{1,\ldots,2^w-1\}$; let $q$ and $i$ be any two elements in $S$. Then there is exactly one value $z \in S$ such that $zq \bmod 2^w = i$.*

*Proof.* Since the number of choices for $z$ and $i$ is the same, it is sufficient to prove that there is *at most* one value $z \in S$ that satisfies $zq \bmod 2^w = i$.

Suppose, for the sake of contradiction, that there are two such values $z$ and $z'$, with $z > z'$. Then

$$zq \bmod 2^w = z'q \bmod 2^w = i$$

So

$$(z-z')q \bmod 2^w = 0$$

But this means that

$$(z-z')q = k2^w \tag{5.1}$$

for some integer $k$. Thinking in terms of binary numbers, we have

$$(z-z')q = k \cdot (1,\underbrace{0,\ldots,0}_{w})_2 \ ,$$

so that the $w$ trailing bits in the binary representation of $(z-z')q$ are all 0's.

Furthermore $k \neq 0$, since $q \neq 0$ and $z - z' \neq 0$. Since $q$ is odd, it has no trailing 0's in its binary representation:

$$q = (\star,\ldots,\star,1)_2 \ .$$

Since $|z-z'| < 2^w$, $z-z'$ has fewer than $w$ trailing 0's in its binary representation:

$$z - z' = (\star,\ldots,\star,1,\underbrace{0,\ldots,0}_{<w})_2 \ .$$

Therefore, the product $(z-z')q$ has fewer than $w$ trailing 0's in its binary representation:

$$(z-z')q = (\star,\cdots,\star,1,\underbrace{0,\ldots,0}_{<w})_2 \ .$$

Therefore $(z - z')q$ cannot satisfy (5.1), yielding a contradiction and completing the proof. □

The utility of Lemma 5.3 comes from the following observation: If $z$ is chosen uniformly at random from $S$, then $zt$ is uniformly distributed over $S$. In the following proof, it helps to think of the binary representation of $z$, which consists of $w-1$ random bits followed by a 1.

*Proof of Lemma 5.1.* First we note that the condition $\mathtt{hash}(x) = \mathtt{hash}(y)$ is equivalent to the statement "the highest-order $d$ bits of $zx \bmod 2^w$ and the highest-order $d$ bits of $zy \bmod 2^w$ are the same." A necessary condition of that statement is that the highest-order $d$ bits in the binary representation of $z(x-y) \bmod 2^w$ are either all 0's or all 1's. That is,

$$z(x-y) \bmod 2^w = (\underbrace{0,\ldots,0}_{d},\underbrace{\star,\ldots,\star}_{w-d})_2 \tag{5.2}$$

when $zx \bmod 2^w > zy \bmod 2^w$ or

$$z(x-y) \bmod 2^w = (\underbrace{1,\ldots,1}_{d},\underbrace{\star,\ldots,\star}_{w-d})_2 \ . \tag{5.3}$$

when $zx \bmod 2^w < zy \bmod 2^w$. Therefore, we only have to bound the probability that $z(x-y) \bmod 2^w$ looks like (5.2) or (5.3).

Let $q$ be the unique odd integer such that $(x-y) \bmod 2^w = q2^r$ for some integer $r \geq 0$. By Lemma 5.3, the binary representation of $zq \bmod 2^w$ has $w-1$ random bits, followed by a 1:

$$zq \bmod 2^w = (\underbrace{b_{w-1},\ldots,b_1}_{w-1},1)_2$$

Therefore, the binary representation of $z(x-y) \bmod 2^w = zq2^r \bmod 2^w$ has $w-r-1$ random bits, followed by a 1, followed by $r$ 0's:

$$z(x-y) \bmod 2^w = zq2^r \bmod 2^w = (\underbrace{b_{w-r-1},\ldots,b_1}_{w-r-1},1,\underbrace{0,0,\ldots,0}_{r})_2$$

We can now finish the proof: If $r > w-d$, then the $d$ higher order bits of $z(x-y) \bmod 2^w$ contain both 0's and 1's, so the probability that $z(x-$

y) mod $2^w$ looks like (5.2) or (5.3) is 0. If r = w − d, then the probability of looking like (5.2) is 0, but the probability of looking like (5.3) is $1/2^{d-1} = 2/2^d$ (since we must have $b_1, \ldots, b_{d-1} = 1, \ldots, 1$). If $r < w - d$, then we must have $b_{w-r-1}, \ldots, b_{w-r-d} = 0, \ldots, 0$ or $b_{w-r-1}, \ldots, b_{w-r-d} = 1, \ldots, 1$. The probability of each of these cases is $1/2^d$ and they are mutually exclusive, so the probability of either of these cases is $2/2^d$. This completes the proof. □

### 5.1.2 Summary

The following theorem summarizes the performance of a ChainedHashTable data structure:

**Theorem 5.1.** *A ChainedHashTable implements the USet interface. Ignoring the cost of calls to* grow(), *a ChainedHashTable supports the operations* add(x), remove(x), *and* find(x) *in $O(1)$ expected time per operation.*

*Furthermore, beginning with an empty ChainedHashTable, any sequence of $m$* add(x) *and* remove(x) *operations results in a total of $O(m)$ time spent during all calls to* grow().

## 5.2 LinearHashTable: Linear Probing

The ChainedHashTable data structure uses an array of lists, where the ith list stores all elements x such that hash(x) = i. An alternative, called *open addressing* is to store the elements directly in an array, t, with each array location in t storing at most one value. This approach is taken by the LinearHashTable described in this section. In some places, this data structure is described as *open addressing with linear probing*.

The main idea behind a LinearHashTable is that we would, ideally, like to store the element x with hash value i = hash(x) in the table location t[i]. If we cannot do this (because some element is already stored there) then we try to store it at location t[(i + 1) mod t.length]; if that's not possible, then we try t[(i + 2) mod t.length], and so on, until we find a place for x.

There are three types of entries stored in t:

1. data values: actual values in the USet that we are representing;
2. null values: at array locations where no data has ever been stored; and
3. del values: at array locations where data was once stored but that has since been deleted.

In addition to the counter, n, that keeps track of the number of elements in the LinearHashTable, a counter, q, keeps track of the number of elements of Types 1 and 3. That is, q is equal to n plus the number of del values in t. To make this work efficiently, we need t to be considerably larger than q, so that there are lots of null values in t. The operations on a LinearHashTable therefore maintain the invariant that $t.length \geq 2q$.

To summarize, a LinearHashTable contains an array, t, that stores data elements, and integers n and q that keep track of the number of data elements and non-null values of t, respectively. Because many hash functions only work for table sizes that are a power of 2, we also keep an integer d and maintain the invariant that $t.length = 2^d$.

LinearHashTable

```
T[] t;   // the table
int n;   // the size
int d;   // t.length = 2^d
int q;   // number of non-null entries in t
```

The find(x) operation in a LinearHashTable is simple. We start at array entry t[i] where i = hash(x) and search entries t[i], t[(i + 1) mod t.length], t[(i + 2) mod t.length], and so on, until we find an index i′ such that, either, t[i′] = x, or t[i′] = null. In the former case we return t[i′]. In the latter case, we conclude that x is not contained in the hash table and return null.

LinearHashTable

```
T find(T x) {
  int i = hash(x);
  while (t[i] != null) {
    if (t[i] != del && x.equals(t[i])) return t[i];
    i = (i == t.length-1) ? 0 : i + 1; // increment i
```

```
  }
  return null;
}
```

The add(x) operation is also fairly easy to implement. After checking that x is not already stored in the table (using find(x)), we search t[i], t[(i+1) mod t.length], t[(i+2) mod t.length], and so on, until we find a null or del and store x at that location, increment n, and q, if appropriate.

LinearHashTable

```
boolean add(T x) {
  if (find(x) != null) return false;
  if (2*(q+1) > t.length) resize(); // max 50% occupancy
  int i = hash(x);
  while (t[i] != null && t[i] != del)
    i = (i == t.length-1) ? 0 : i + 1; // increment i
  if (t[i] == null) q++;
  n++;
  t[i] = x;
  return true;
}
```

By now, the implementation of the remove(x) operation should be obvious. We search t[i], t[(i + 1) mod t.length], t[(i + 2) mod t.length], and so on until we find an index i′ such that t[i′] = x or t[i′] = null. In the former case, we set t[i′] = del and return true. In the latter case we conclude that x was not stored in the table (and therefore cannot be deleted) and return false.

LinearHashTable

```
T remove(T x) {
  int i = hash(x);
  while (t[i] != null) {
    T y = t[i];
    if (y != del && x.equals(y)) {
      t[i] = del;
      n--;
      if (8*n < t.length) resize(); // min 12.5% occupancy
      return y;
```

```
    }
    i = (i == t.length-1) ? 0 : i + 1;  // increment i
  }
  return null;
}
```

The correctness of the `find(x)`, `add(x)`, and `remove(x)` methods is easy to verify, though it relies on the use of `del` values. Notice that none of these operations ever sets a non-`null` entry to `null`. Therefore, when we reach an index `i`′ such that `t[i']` = `null`, this is a proof that the element, `x`, that we are searching for is not stored in the table; `t[i']` has always been `null`, so there is no reason that a previous `add(x)` operation would have proceeded beyond index `i`′.

The `resize()` method is called by `add(x)` when the number of non-`null` entries exceeds `t.length`/2 or by `remove(x)` when the number of data entries is less than `t.length`/8. The `resize()` method works like the `resize()` methods in other array-based data structures. We find the smallest non-negative integer `d` such that $2^d \geq 3n$. We reallocate the array `t` so that it has size $2^d$, and then we insert all the elements in the old version of `t` into the newly-resized copy of `t`. While doing this, we reset `q` equal to `n` since the newly-allocated `t` contains no `del` values.

LinearHashTable

```
void resize() {
  d = 1;
  while ((1<<d) < 3*n) d++;
  T[] told = t;
  t = newArray(1<<d);
  q = n;
  // insert everything from told
  for (int k = 0; k < told.length; k++) {
    if (told[k] != null && told[k] != del) {
      int i = hash(told[k]);
      while (t[i] != null)
        i = (i == t.length-1) ? 0 : i + 1;
      t[i] = told[k];
    }
  }
```

```
}
```

### 5.2.1 Analysis of Linear Probing

Notice that each operation, `add(x)`, `remove(x)`, or `find(x)`, finishes as soon as (or before) it discovers the first `null` entry in `t`. The intuition behind the analysis of linear probing is that, since at least half the elements in `t` are equal to `null`, an operation should not take long to complete because it will very quickly come across a `null` entry. We shouldn't rely too heavily on this intuition, though, because it would lead us to (the incorrect) conclusion that the expected number of locations in `t` examined by an operation is at most 2.

For the rest of this section, we will assume that all hash values are independently and uniformly distributed in $\{0,\ldots,\texttt{t.length}-1\}$. This is not a realistic assumption, but it will make it possible for us to analyze linear probing. Later in this section we will describe a method, called tabulation hashing, that produces a hash function that is "good enough" for linear probing. We will also assume that all indices into the positions of `t` are taken modulo `t.length`, so that `t[i]` is really a shorthand for `t[i` mod `t.length]`.

We say that a *run of length $k$ that starts at* `i` occurs when all the table entries $\texttt{t}[\texttt{i}], \texttt{t}[\texttt{i}+1], \ldots, \texttt{t}[\texttt{i}+k-1]$ are non-`null` and $\texttt{t}[\texttt{i}-1] = \texttt{t}[\texttt{i}+k] = \texttt{null}$. The number of non-`null` elements of `t` is exactly `q` and the `add(x)` method ensures that, at all times, $\texttt{q} \le \texttt{t.length}/2$. There are `q` elements $\texttt{x}_1,\ldots,\texttt{x}_\texttt{q}$ that have been inserted into `t` since the last `rebuild()` operation. By our assumption, each of these has a hash value, $\texttt{hash}(\texttt{x}_j)$, that is uniform and independent of the rest. With this setup, we can prove the main lemma required to analyze linear probing.

**Lemma 5.4.** *Fix a value* $\texttt{i} \in \{0,\ldots,\texttt{t.length}-1\}$. *Then the probability that a run of length $k$ starts at* `i` *is $O(c^k)$ for some constant $0 < c < 1$.*

*Proof.* If a run of length $k$ starts at `i`, then there are exactly $k$ elements $\texttt{x}_j$ such that $\texttt{hash}(\texttt{x}_j) \in \{\texttt{i},\ldots,\texttt{i}+k-1\}$. The probability that this occurs is exactly

$$p_k = \binom{\texttt{q}}{k}\left(\frac{k}{\texttt{t.length}}\right)^k\left(\frac{\texttt{t.length}-k}{\texttt{t.length}}\right)^{\texttt{q}-k} ,$$

since, for each choice of $k$ elements, these $k$ elements must hash to one of the $k$ locations and the remaining $q-k$ elements must hash to the other $t.length-k$ table locations.[1]

In the following derivation we will cheat a little and replace $r!$ with $(r/e)^r$. Stirling's Approximation (Section 1.3.2) shows that this is only a factor of $O(\sqrt{r})$ from the truth. This is just done to make the derivation simpler; Exercise 5.4 asks the reader to redo the calculation more rigorously using Stirling's Approximation in its entirety.

The value of $p_k$ is maximized when $t.length$ is minimum, and the data structure maintains the invariant that $t.length \geq 2q$, so

$$\begin{aligned}
p_k &\leq \binom{q}{k}\left(\frac{k}{2q}\right)^k\left(\frac{2q-k}{2q}\right)^{q-k} \\
&= \left(\frac{q!}{(q-k)!k!}\right)\left(\frac{k}{2q}\right)^k\left(\frac{2q-k}{2q}\right)^{q-k} \\
&\approx \left(\frac{q^q}{(q-k)^{q-k}k^k}\right)\left(\frac{k}{2q}\right)^k\left(\frac{2q-k}{2q}\right)^{q-k} && \text{[Stirling's approximation]} \\
&= \left(\frac{q^k q^{q-k}}{(q-k)^{q-k}k^k}\right)\left(\frac{k}{2q}\right)^k\left(\frac{2q-k}{2q}\right)^{q-k} \\
&= \left(\frac{qk}{2qk}\right)^k\left(\frac{q(2q-k)}{2q(q-k)}\right)^{q-k} \\
&= \left(\frac{1}{2}\right)^k\left(\frac{(2q-k)}{2(q-k)}\right)^{q-k} \\
&= \left(\frac{1}{2}\right)^k\left(1+\frac{k}{2(q-k)}\right)^{q-k} \\
&\leq \left(\frac{\sqrt{e}}{2}\right)^k .
\end{aligned}$$

(In the last step, we use the inequality $(1+1/x)^x \leq e$, which holds for all $x > 0$.) Since $\sqrt{e}/2 < 0.824360636 < 1$, this completes the proof. □

Using Lemma 5.4 to prove upper-bounds on the expected running time of `find(x)`, `add(x)`, and `remove(x)` is now fairly straightforward. Consider the simplest case, where we execute `find(x)` for some value `x` that

[1] Note that $p_k$ is greater than the probability that a run of length $k$ starts at `i`, since the definition of $p_k$ does not include the requirement $t[i-1] = t[i+k] = null$.

has never been stored in the `LinearHashTable`. In this case, `i` = `hash(x)` is a random value in $\{0,\ldots,$ `t.length` $-1\}$ independent of the contents of `t`. If `i` is part of a run of length $k$, then the time it takes to execute the `find(x)` operation is at most $O(1+k)$. Thus, the expected running time can be upper-bounded by

$$O\left(1+\left(\frac{1}{\texttt{t.length}}\right)\sum_{i=1}^{\texttt{t.length}}\sum_{k=0}^{\infty} k\Pr\{\texttt{i} \text{ is part of a run of length } k\}\right) .$$

Note that each run of length $k$ contributes to the inner sum $k$ times for a total contribution of $k^2$, so the above sum can be rewritten as

$$\begin{aligned}
&O\left(1+\left(\frac{1}{\texttt{t.length}}\right)\sum_{i=1}^{\texttt{t.length}}\sum_{k=0}^{\infty} k^2\Pr\{\texttt{i} \text{ starts a run of length } k\}\right)\\
&\le O\left(1+\left(\frac{1}{\texttt{t.length}}\right)\sum_{i=1}^{\texttt{t.length}}\sum_{k=0}^{\infty} k^2 p_k\right)\\
&= O\left(1+\sum_{k=0}^{\infty} k^2 p_k\right)\\
&= O\left(1+\sum_{k=0}^{\infty} k^2\cdot O(c^k)\right)\\
&= O(1) .
\end{aligned}$$

The last step in this derivation comes from the fact that $\sum_{k=0}^{\infty} k^2\cdot O(c^k)$ is an exponentially decreasing series.[2] Therefore, we conclude that the expected running time of the `find(x)` operation for a value `x` that is not contained in a `LinearHashTable` is $O(1)$.

If we ignore the cost of the `resize()` operation, then the above analysis gives us all we need to analyze the cost of operations on a `LinearHashTable`.

First of all, the analysis of `find(x)` given above applies to the `add(x)` operation when `x` is not contained in the table. To analyze the `find(x)` operation when `x` is contained in the table, we need only note that this

[2] In the terminology of many calculus texts, this sum passes the ratio test: There exists a positive integer $k_0$ such that, for all $k \ge k_0$, $\frac{(k+1)^2 c^{k+1}}{k^2 c^k} < 1$.

is the same as the cost of the add(x) operation that previously added x to the table. Finally, the cost of a remove(x) operation is the same as the cost of a find(x) operation.

In summary, if we ignore the cost of calls to resize(), all operations on a LinearHashTable run in $O(1)$ expected time. Accounting for the cost of resize can be done using the same type of amortized analysis performed for the ArrayStack data structure in Section 2.1.

### 5.2.2 Summary

The following theorem summarizes the performance of the LinearHashTable data structure:

**Theorem 5.2.** *A LinearHashTable implements the USet interface. Ignoring the cost of calls to* resize(), *a LinearHashTable supports the operations* add(x), remove(x), *and* find(x) *in $O(1)$ expected time per operation.*

*Furthermore, beginning with an empty LinearHashTable, any sequence of $m$* add(x) *and* remove(x) *operations results in a total of $O(m)$ time spent during all calls to* resize().

### 5.2.3 Tabulation Hashing

While analyzing the LinearHashTable structure, we made a very strong assumption: That for any set of elements, $\{x_1,\ldots,x_n\}$, the hash values $\texttt{hash}(x_1),\ldots,\texttt{hash}(x_n)$ are independently and uniformly distributed over the set $\{0,\ldots,\texttt{t.length}-1\}$. One way to achieve this is to store a giant array, tab, of length $2^w$, where each entry is a random w-bit integer, independent of all the other entries. In this way, we could implement hash(x) by extracting a d-bit integer from tab[x.hashCode()]:

LinearHashTable

```
int idealHash(T x) {
  return tab[x.hashCode() >>> w-d];
}
```

Unfortunately, storing an array of size $2^w$ is prohibitive in terms of memory usage. The approach used by *tabulation hashing* is to, instead,

treat w-bit integers as being comprised of w/r integers, each having only r bits. In this way, tabulation hashing only needs w/r arrays each of length $2^r$. All the entries in these arrays are independent w-bit integers. To obtain the value of hash(x) we split x.hashCode() up into w/r r-bit integers and use these as indices into these arrays. We then combine all these values with the bitwise exclusive-or operator to obtain hash(x). The following code shows how this works when w = 32 and r = 4:

LinearHashTable
```
int hash(T x) {
  int h = x.hashCode();
  return (tab[0][h&0xff]
          ^ tab[1][(h>>>8)&0xff]
          ^ tab[2][(h>>>16)&0xff]
          ^ tab[3][(h>>>24)&0xff])
         >>> (w-d);
}
```

In this case, tab is a two-dimensional array with four columns and $2^{32/4} = 256$ rows.

One can easily verify that, for any x, hash(x) is uniformly distributed over $\{0,\ldots,2^d-1\}$. With a little work, one can even verify that any pair of values have independent hash values. This implies tabulation hashing could be used in place of multiplicative hashing for the ChainedHashTable implementation.

However, it is not true that any set of n distinct values gives a set of n independent hash values. Nevertheless, when tabulation hashing is used, the bound of Theorem 5.2 still holds. References for this are provided at the end of this chapter.

## 5.3 Hash Codes

The hash tables discussed in the previous section are used to associate data with integer keys consisting of w bits. In many cases, we have keys that are not integers. They may be strings, objects, arrays, or other compound structures. To use hash tables for these types of data, we must

map these data types to w-bit hash codes. Hash code mappings should have the following properties:

1. If x and y are equal, then x.hashCode() and y.hashCode() are equal.
2. If x and y are not equal, then the probability that x.hashCode() = y.hashCode() should be small (close to $1/2^w$).

The first property ensures that if we store x in a hash table and later look up a value y equal to x, then we will find x—as we should. The second property minimizes the loss from converting our objects to integers. It ensures that unequal objects usually have different hash codes and so are likely to be stored at different locations in our hash table.

### 5.3.1 Hash Codes for Primitive Data Types

Small primitive data types like `char`, `byte`, `int`, and `float` are usually easy to find hash codes for. These data types always have a binary representation and this binary representation usually consists of w or fewer bits. (For example, in Java, `byte` is an 8-bit type and `float` is a 32-bit type.) In these cases, we just treat these bits as the representation of an integer in the range $\{0,\ldots,2^w-1\}$. If two values are different, they get different hash codes. If they are the same, they get the same hash code.

A few primitive data types are made up of more than w bits, usually $cw$ bits for some constant integer $c$. (Java's `long` and `double` types are examples of this with $c = 2$.) These data types can be treated as compound objects made of $c$ parts, as described in the next section.

### 5.3.2 Hash Codes for Compound Objects

For a compound object, we want to create a hash code by combining the individual hash codes of the object's constituent parts. This is not as easy as it sounds. Although one can find many hacks for this (for example, combining the hash codes with bitwise exclusive-or operations), many of these hacks turn out to be easy to foil (see Exercises 5.7–5.9). However, if one is willing to do arithmetic with 2w bits of precision, then there are simple and robust methods available. Suppose we have an object made

up of several parts $P_0,\ldots,P_{r-1}$ whose hash codes are $x_0,\ldots,x_{r-1}$. Then we can choose mutually independent random $w$-bit integers $z_0,\ldots,z_{r-1}$ and a random $2w$-bit odd integer $z$ and compute a hash code for our object with

$$h(x_0,\ldots,x_{r-1}) = \left(\left(z\sum_{i=0}^{r-1} z_i x_i\right) \bmod 2^{2w}\right) \operatorname{div} 2^w \ .$$

Note that this hash code has a final step (multiplying by $z$ and dividing by $2^w$) that uses the multiplicative hash function from Section 5.1.1 to take the $2w$-bit intermediate result and reduce it to a $w$-bit final result. Here is an example of this method applied to a simple compound object with three parts `x0`, `x1`, and `x2`:

Point3D

```
int hashCode() {
  // random numbers from rand.org
  long[] z = {0x2058cc50L, 0xcb19137eL, 0x2cb6b6fdL};
  long zz = 0xbea0107e5067d19dL;

  // convert (unsigned) hashcodes to long
  long h0 = x0.hashCode() & ((1L<<32)-1);
  long h1 = x1.hashCode() & ((1L<<32)-1);
  long h2 = x2.hashCode() & ((1L<<32)-1);

  return (int)(((z[0]*h0 + z[1]*h1 + z[2]*h2)*zz)
               >>> 32);
}
```

The following theorem shows that, in addition to being straightforward to implement, this method is provably good:

**Theorem 5.3.** *Let $x_0,\ldots,x_{r-1}$ and $y_0,\ldots,y_{r-1}$ each be sequences of $w$ bit integers in $\{0,\ldots,2^w-1\}$ and assume $x_i \neq y_i$ for at least one index $i \in \{0,\ldots,r-1\}$. Then*

$$\Pr\{h(x_0,\ldots,x_{r-1}) = h(y_0,\ldots,y_{r-1})\} \leq 3/2^w \ .$$

*Proof.* We will first ignore the final multiplicative hashing step and see how that step contributes later. Define:

$$h'(x_0,\ldots,x_{r-1}) = \left(\sum_{j=0}^{r-1} z_j x_j\right) \bmod 2^{2w} \ .$$

Suppose that $h'(x_0,\ldots,x_{r-1}) = h'(y_0,\ldots,y_{r-1})$. We can rewrite this as:

$$z_i(x_i - y_i) \bmod 2^{2w} = t \tag{5.4}$$

where

$$t = \left( \sum_{j=0}^{i-1} z_j(y_j - x_j) + \sum_{j=i+1}^{r-1} z_j(y_j - x_j) \right) \bmod 2^{2w}$$

If we assume, without loss of generality that $x_i > y_i$, then (5.4) becomes

$$z_i(x_i - y_i) = t \ , \tag{5.5}$$

since each of $z_i$ and $(x_i - y_i)$ is at most $2^w - 1$, so their product is at most $2^{2w} - 2^{w+1} + 1 < 2^{2w} - 1$. By assumption, $x_i - y_i \neq 0$, so (5.5) has at most one solution in $z_i$. Therefore, since $z_i$ and $t$ are independent ($z_0,\ldots,z_{r-1}$ are mutually independent), the probability that we select $z_i$ so that $h'(x_0,\ldots,x_{r-1}) = h'(y_0,\ldots,y_{r-1})$ is at most $1/2^w$.

The final step of the hash function is to apply multiplicative hashing to reduce our $2w$-bit intermediate result $h'(x_0,\ldots,x_{r-1})$ to a $w$-bit final result $h(x_0,\ldots,x_{r-1})$. By Theorem 5.3, if $h'(x_0,\ldots,x_{r-1}) \neq h'(y_0,\ldots,y_{r-1})$, then $\Pr\{h(x_0,\ldots,x_{r-1}) = h(y_0,\ldots,y_{r-1})\} \leq 2/2^w$.

To summarize,

$$\begin{aligned}
&\Pr\left\{ \begin{array}{l} h(x_0,\ldots,x_{r-1}) \\ \quad = h(y_0,\ldots,y_{r-1}) \end{array} \right\} \\
&= \Pr\left\{ \begin{array}{l} h'(x_0,\ldots,x_{r-1}) = h'(y_0,\ldots,y_{r-1}) \text{ or} \\ h'(x_0,\ldots,x_{r-1}) \neq h'(y_0,\ldots,y_{r-1}) \\ \quad \text{and } z h'(x_0,\ldots,x_{r-1}) \operatorname{div} 2^w = z h'(y_0,\ldots,y_{r-1}) \operatorname{div} 2^w \end{array} \right\} \\
&\leq 1/2^w + 2/2^w = 3/2^w \ .
\end{aligned}$$

□

### 5.3.3 Hash Codes for Arrays and Strings

The method from the previous section works well for objects that have a fixed, constant, number of components. However, it breaks down when we want to use it with objects that have a variable number of components, since it requires a random $w$-bit integer $z_i$ for each component. We could use a pseudorandom sequence to generate as many $z_i$'s as we need, but then the $z_i$'s are not mutually independent, and it becomes difficult to

prove that the pseudorandom numbers don't interact badly with the hash function we are using. In particular, the values of $t$ and $z_i$ in the proof of Theorem 5.3 are no longer independent.

A more rigorous approach is to base our hash codes on polynomials over prime fields; these are just regular polynomials that are evaluated modulo some prime number, $p$. This method is based on the following theorem, which says that polynomials over prime fields behave pretty-much like usual polynomials:

**Theorem 5.4.** *Let $p$ be a prime number, and let $f(z) = x_0 z^0 + x_1 z^1 + \cdots + x_{r-1} z^{r-1}$ be a non-trivial polynomial with coefficients $x_i \in \{0,\ldots,p-1\}$. Then the equation $f(z) \bmod p = 0$ has at most $r-1$ solutions for $z \in \{0,\ldots,p-1\}$.*

To use Theorem 5.4, we hash a sequence of integers $x_0,\ldots,x_{r-1}$ with each $x_i \in \{0,\ldots,p-2\}$ using a random integer $z \in \{0,\ldots,p-1\}$ via the formula

$$h(x_0,\ldots,x_{r-1}) = \left(x_0 z^0 + \cdots + x_{r-1} z^{r-1} + (p-1)z^r\right) \bmod p \ .$$

Note the extra $(p-1)z^r$ term at the end of the formula. It helps to think of $(p-1)$ as the last element, $x_r$, in the sequence $x_0,\ldots,x_r$. Note that this element differs from every other element in the sequence (each of which is in the set $\{0,\ldots,p-2\}$). We can think of $p-1$ as an end-of-sequence marker.

The following theorem, which considers the case of two sequences of the same length, shows that this hash function gives a good return for the small amount of randomization needed to choose $z$:

**Theorem 5.5.** *Let $p > 2^w + 1$ be a prime, let $x_0,\ldots,x_{r-1}$ and $y_0,\ldots,y_{r-1}$ each be sequences of $w$-bit integers in $\{0,\ldots,2^w-1\}$, and assume $x_i \neq y_i$ for at least one index $i \in \{0,\ldots,r-1\}$. Then*

$$\Pr\{h(x_0,\ldots,x_{r-1}) = h(y_0,\ldots,y_{r-1})\} \leq (r-1)/p\} \ .$$

*Proof.* The equation $h(x_0,\ldots,x_{r-1}) = h(y_0,\ldots,y_{r-1})$ can be rewritten as

$$\left((x_0 - y_0)z^0 + \cdots + (x_{r-1} - y_{r-1})z^{r-1}\right) \bmod p = 0. \tag{5.6}$$

Since $x_i \neq y_i$, this polynomial is non-trivial. Therefore, by Theorem 5.4, it has at most $r-1$ solutions in $z$. The probability that we pick $z$ to be one of these solutions is therefore at most $(r-1)/p$. □

Note that this hash function also deals with the case in which two sequences have different lengths, even when one of the sequences is a prefix of the other. This is because this function effectively hashes the infinite sequence

$$x_0, \ldots, x_{r-1}, p-1, 0, 0, \ldots \ .$$

This guarantees that if we have two sequences of length $r$ and $r'$ with $r > r'$, then these two sequences differ at index $i = r$. In this case, (5.6) becomes

$$\left( \sum_{i=0}^{i=r'-1} (x_i - y_i) z^i + (x_{r'} - p + 1) z^{r'} + \sum_{i=r'+1}^{i=r-1} x_i z^i + (p-1) z^r \right) \bmod p = 0 \ ,$$

which, by Theorem 5.4, has at most $r$ solutions in $z$. This combined with Theorem 5.5 suffice to prove the following more general theorem:

**Theorem 5.6.** *Let $p > 2^w + 1$ be a prime, let $x_0, \ldots, x_{r-1}$ and $y_0, \ldots, y_{r'-1}$ be distinct sequences of $w$-bit integers in $\{0, \ldots, 2^w - 1\}$. Then*

$$\Pr\{h(x_0, \ldots, x_{r-1}) = h(y_0, \ldots, y_{r-1})\} \leq \max\{r, r'\}/p \ .$$

The following example code shows how this hash function is applied to an object that contains an array, `x`, of values:

GeomVector

```
int hashCode() {
  long p = (1L<<32)-5;   // prime: 2^32 - 5
  long z = 0x64b6055aL;  // 32 bits from random.org
  int z2 = 0x5067d19d;   // random odd 32 bit number
  long s = 0;
  long zi = 1;
  for (int i = 0; i < x.length; i++) {
    // reduce to 31 bits
    long xi = (x[i].hashCode() * z2) >>> 1;
    s = (s + zi * xi) % p;
    zi = (zi * z) % p;
  }
  s = (s + zi * (p-1)) % p;
  return (int)s;
}
```

The preceding code sacrifices some collision probability for implementation convenience. In particular, it applies the multiplicative hash function from Section 5.1.1, with d = 31 to reduce x[i].hashCode() to a 31-bit value. This is so that the additions and multiplications that are done modulo the prime p = $2^{32} - 5$ can be carried out using unsigned 63-bit arithmetic. Thus the probability of two different sequences, the longer of which has length $r$, having the same hash code is at most

$$2/2^{31} + r/(2^{32} - 5)$$

rather than the $r/(2^{32} - 5)$ specified in Theorem 5.6.

## 5.4 Discussion and Exercises

Hash tables and hash codes represent an enormous and active field of research that is just touched upon in this chapter. The online Bibliography on Hashing [10] contains nearly 2000 entries.

A variety of different hash table implementations exist. The one described in Section 5.1 is known as *hashing with chaining* (each array entry contains a chain (List) of elements). Hashing with chaining dates back to an internal IBM memorandum authored by H. P. Luhn and dated January 1953. This memorandum also seems to be one of the earliest references to linked lists.

An alternative to hashing with chaining is that used by *open addressing* schemes, where all data is stored directly in an array. These schemes include the LinearHashTable structure of Section 5.2. This idea was also proposed, independently, by a group at IBM in the 1950s. Open addressing schemes must deal with the problem of *collision resolution*: the case where two values hash to the same array location. Different strategies exist for collision resolution; these provide different performance guarantees and often require more sophisticated hash functions than the ones described here.

Yet another category of hash table implementations are the so-called *perfect hashing* methods. These are methods in which find(x) operations take $O(1)$ time in the worst-case. For static data sets, this can be accomplished by finding *perfect hash functions* for the data; these are functions

that map each piece of data to a unique array location. For data that changes over time, perfect hashing methods include *FKS two-level hash tables* [31, 24] and *cuckoo hashing* [57].

The hash functions presented in this chapter are probably among the most practical methods currently known that can be proven to work well for any set of data. Other provably good methods date back to the pioneering work of Carter and Wegman who introduced the notion of *universal hashing* and described several hash functions for different scenarios [14]. Tabulation hashing, described in Section 5.2.3, is due to Carter and Wegman [14], but its analysis, when applied to linear probing (and several other hash table schemes) is due to Pătraşcu and Thorup [60].

The idea of *multiplicative hashing* is very old and seems to be part of the hashing folklore [48, Section 6.4]. However, the idea of choosing the multiplier $z$ to be a random *odd* number, and the analysis in Section 5.1.1 is due to Dietzfelbinger *et al.* [23]. This version of multiplicative hashing is one of the simplest, but its collision probability of $2/2^d$ is a factor of two larger than what one could expect with a random function from $2^w \to 2^d$. The *multiply-add hashing* method uses the function

$$h(x) = ((zx + b) \bmod 2^{2w}) \operatorname{div} 2^{2w-d}$$

where $z$ and $b$ are each randomly chosen from $\{0, \ldots, 2^{2w}-1\}$. Multiply-add hashing has a collision probability of only $1/2^d$ [21], but requires $2w$-bit precision arithmetic.

There are a number of methods of obtaining hash codes from fixed-length sequences of $w$-bit integers. One particularly fast method [11] is the function

$$h(x_0, \ldots, x_{r-1}) = \left(\sum_{i=0}^{r/2-1} ((x_{2i} + a_{2i}) \bmod 2^w)((x_{2i+1} + a_{2i+1}) \bmod 2^w)\right) \bmod 2^{2w}$$

where $r$ is even and $a_0, \ldots, a_{r-1}$ are randomly chosen from $\{0, \ldots, 2^w\}$. This yields a $2w$-bit hash code that has collision probability $1/2^w$. This can be reduced to a $w$-bit hash code using multiplicative (or multiply-add) hashing. This method is fast because it requires only $r/2$ $2w$-bit multiplications whereas the method described in Section 5.3.2 requires $r$ multiplications. (The mod operations occur implicitly by using $w$ and $2w$-bit arithmetic for the additions and multiplications, respectively.)

The method from Section 5.3.3 of using polynomials over prime fields to hash variable-length arrays and strings is due to Dietzfelbinger *et al.* [22]. Due to its use of the mod operator which relies on a costly machine instruction, it is, unfortunately, not very fast. Some variants of this method choose the prime p to be one of the form $2^w - 1$, in which case the mod operator can be replaced with addition (+) and bitwise-and (&) operations [47, Section 3.6]. Another option is to apply one of the fast methods for fixed-length strings to blocks of length $c$ for some constant $c > 1$ and then apply the prime field method to the resulting sequence of $\lceil r/c \rceil$ hash codes.

**Exercise 5.1.** A certain university assigns each of its students student numbers the first time they register for any course. These numbers are sequential integers that started at 0 many years ago and are now in the millions. Suppose we have a class of one hundred first year students and we want to assign them hash codes based on their student numbers. Does it make more sense to use the first two digits or the last two digits of their student number? Justify your answer.

**Exercise 5.2.** Consider the hashing scheme in Section 5.1.1, and suppose $n = 2^d$ and $d \leq w/2$.

1. Show that, for any choice of the muliplier, z, there exists n values that all have the same hash code. (Hint: This is easy, and doesn't require any number theory.)

2. Given the multiplier, z, describe n values that all have the same hash code. (Hint: This is harder, and requires some basic number theory.)

**Exercise 5.3.** Prove that the bound $2/2^d$ in Lemma 5.1 is the best possible bound by showing that, if $x = 2^{w-d-2}$ and $y = 3x$, then $\Pr\{\text{hash}(x) = \text{hash}(y)\} = 2/2^d$. (Hint look at the binary representations of zx and z3x and use the fact that $z3x = zx + 2zx$.)

**Exercise 5.4.** Reprove Lemma 5.4 using the full version of Stirling's Approximation given in Section 1.3.2.

**Exercise 5.5.** Consider the following simplified version of the code for adding an element x to a `LinearHashTable`, which simply stores x in the

first `null` array entry it finds. Explain why this could be very slow by giving an example of a sequence of $O(n)$ `add(x)`, `remove(x)`, and `find(x)` operations that would take on the order of $n^2$ time to execute.

LinearHashTable

```
boolean addSlow(T x) {
  if (2*(q+1) > t.length) resize(); // max 50% occupancy
  int i = hash(x);
  while (t[i] != null) {
    if (t[i] != del && x.equals(t[i])) return false;
    i = (i == t.length-1) ? 0 : i + 1; // increment i
  }
  t[i] = x;
  n++; q++;
  return true;
}
```

**Exercise 5.6.** Early versions of the Java `hashCode()` method for the `String` class worked by not using all of the characters found in long strings. For example, for a sixteen character string, the hash code was computed using only the eight even-indexed characters. Explain why this was a very bad idea by giving an example of large set of strings that all have the same hash code.

**Exercise 5.7.** Suppose you have an object made up of two $w$-bit integers, $x$ and $y$. Show why $x \oplus y$ does not make a good hash code for your object. Give an example of a large set of objects that would all have hash code 0.

**Exercise 5.8.** Suppose you have an object made up of two $w$-bit integers, $x$ and $y$. Show why $x + y$ does not make a good hash code for your object. Give an example of a large set of objects that would all have the same hash code.

**Exercise 5.9.** Suppose you have an object made up of two $w$-bit integers, $x$ and $y$. Suppose that the hash code for your object is defined by some deterministic function $h(x, y)$ that produces a single $w$-bit integer. Prove that there exists a large set of objects that have the same hash code.

**Exercise 5.10.** Let $p = 2^w - 1$ for some positive integer $w$. Explain why, for

a positive integer $x$

$$(x \bmod 2^w) + (x \operatorname{div} 2^w) \equiv x \bmod (2^w - 1) \ .$$

(This gives an algorithm for computing $x \bmod (2^w - 1)$ by repeatedly setting

$$\texttt{x} = \texttt{x\&((1<<w)} - 1) + \texttt{x>>>w}$$

until $\texttt{x} \leq 2^w - 1$.)

**Exercise 5.11.** Find some commonly used hash table implementation such as the (Java Collection Framework `HashMap` or the `HashTable` or `LinearHashTable` implementations in this book, and design a program that stores integers in this data structure so that there are integers, `x`, such that `find(x)` takes linear time. That is, find a set of `n` integers for which there are $c$`n` elements that hash to the same table location.

Depending on how good the implementation is, you may be able to do this just by inspecting the code for the implementation, or you may have to write some code that does trial insertions and searches, timing how long it takes to add and find particular values. (This can be, and has been, used to launch denial of service attacks on web servers [17].)